Robert Dudziński 293103
TKOM – Dokumentacja końcowa
Temat: O3 - niezmienniki pętli

Celem projektu jest stworzenie programu, który w pierwszej kolejności będzie weryfikował poprawność kodu wejściowego napisanego w niżej zdefiniowanym języku, a następnie będzie go optymalizował poprzez wykrywanie niezmienników i przenoszenie ich przed pętlę, w której się znajdują (wraz z obsługą zagnieżdżonych pętli).

WYMAGANIA:

- odczytanie kodu źródłowego z pliku
- analiza leksykalna, składniowa oraz semantyczna kodu wejściowego
- zgłaszanie informacji o pierwszym napotkanym błędzie podczas analiz (wraz z podaniem, w którym miejscu on występuje i prostym opisem przyczyny błędu)
- kod po optymalizacji musi zwracać te same rezultaty jak przed optymalizacją
- wyświetlenie rezultatu na wyjściu standardowym

Język będzie obsługiwał:

- warunki if else
- pętle while, for
- zmienne globalne i lokalne (zmienne lokalne deklarowane w danym bloku przesłaniają zmienne lokalne o tym samym identyfikatorze zadeklarowane w blokach nadrzędnych)
- wyrażenia arytmetyczne i logiczne
- deklaracje zmiennych o stałym typie
- obsługa typów int, float, bool oraz ich jednowymiarowych tablic

Przyjęte założenia:

- całe wejście dla programu musi znajdować się w jednym pliku
- plik zawiera jedną bezargumentową funkcję void main(), która w swojej definicji będzie zawierać cały kod, który będzie poddany analizie (dopuszcza się deklarację zmiennych globalnych)
- nazwa funkcji może być dowolna (musi być zbudowana jak nazwa zwykłej zmiennej) i nie musi być unikalna wobec nazw deklarowanych wcześniej lub później zmiennych
- w kodzie nie ma możliwości korzystania ze wskaźników i operacji na fragmentach pamięci
- w przypadku wyrażeń logicznych wartości typu int oraz float równe 0 będą traktowane jako fałsz, a każda inna wartość będzie traktowana jako prawda
- w przypadku wyrażeń arytmetycznych zmienne typu bool w przypadku fałszu będą traktowane jako 0, natomiast w przeciwnym wypadku będą traktowane jako 1
- przypisanie wartości przy inicjalizacji tablicy skutkuje tym, że każdy jej element będzie miał taką wartość np. „int tab[4] = 2;” sprawi, że każdy element tablicy tab będzie miał wartość równą 2
- jeżeli przy inicjalizacji zmiennej nie przypisano jej wartości to domyślnie będzie przypisywana wartość 0
- język nie będzie obsługiwał prefixowych i postfixowych operatorów tj. „++” oraz „--‘’

Lista zdefiniowanych tokenów:

```
“void”     “(“, “)”   “{“        “}”        “+”        “-“        “*”
“/“        “;”        “if”       “else”     “while”    “for”      “&&”
“||”       “//“       “,”        “<“        “>”        “<=“       “>=“
“==“       “!=“       “=“        “!”        “int”      “float”    “bool
“true”     “false”    “return”   “continue” “break”    “[“        “]”
```

GRAMATYKA

Uwaga: W celu czytelniejszego przedstawienia gramatyki symbol złączenia ‘,’ oznacza zapis ‘, { whiteChar } , ‘ gdzie whiteChar to zbiór białych znaków. Odstępstwem od tej zasady są symbole złączenia użyte przy definiowaniu id, varValue oraz intValue.

```
program = { initVar , “;” } , defFunction ;

defFunction = “void” , id , “(“ , “)” , block ;
block = singleStatement | ( “{“ , { singleStatement | block } , “}” ) ;
singleStatement = (conditionStatement | loopStatement ) |
                  ( ( initVar | return | break | continue | assignVar ) , “;” ) ;

return = “return” ;
break = “break” ;
continue = “continue” ;

conditionStatement = “if” , “(“ , logicalStatement , “)” , block , [“else” , block];
loopStatement = ( “while” , “(“ , logicalStatement , “)” , block ) |
                ( “for” , “(“ , [ assignVar | initVar ] , “;” ,
                  [ logicalStatement ] , “;” , [ assignVar ] , “)” , block ) ;

logicalStatement = andCondition , { “||” , andCondition } ;
andCondition = equalCondition , { “&&” , equalCondition } ;
equalCondition = relationalCondition , { equalOperator , relationalCondition } ;
relationalCondition = logicalParam , { relationalOperator , logicalParam } ;
logicalParam = [ “!” ] ,  ( “(“ , logicalStatement , “)” | expression )  ;

equalOperator = “==“ | “!=“ ;
relationalOperator = “<“ | “<=“ | “>=“ | “>” ;

initVar = varType , var , [ “=“ , expression ] ;
assignVar = var , “=“ , expression ;

expression = ( multiExpression , { addOperator  , multiExpression } ) ;
multiExpression = multiParam , { multiOperator , multiParam } ;
multiParam = [“-“] , (var | varValue | ( “(“ , expression , “)” ) );

multiOperator = “*” | “/” ;
addOperator = “+” | “-“ ;

id = alphaChar , { alphaNumChar } ;
var = id , [ index ] ;
index = “[“ , intValue , “]” ;
```

```
varType = “int” | “float” | “bool” ;
varValue = logicalVal | ( , (
          ( digitChar , [ “.” [ { digitChar } ] ] ) |
          ( nonZeroChar , { digitChar } , [ “.” [ { digitChar } ] ] ) |
          ( “.” { digitChar } )
          ) ) ;
intValue = nonZeroChar , { digitChar } ;

logicalVal = “false” | “true” ;
alphaNumChar = alphaChar | digitChar ;
alphaChar = ? duże i małe znaki alfabetu angielskiego oraz znak ‘_’ ? ;
digitChar = ‘0’ | nonZeroChar ;
nonZeroChar = ‘1’ | ‘2’ | ‘3’ | ‘4’ | ‘5’ | ‘6’ | ‘7’ | ‘8’ | ‘9’ ;
```

BUDOWA PROGRAMU (PODZIAŁ NA MODUŁY):

Obsługa wejścia - InputManager

Oddzielny moduł będzie odpowiedzialny za obsługę ciągu wejściowego. Będzie on posiadał metody do zwracania kolejnych znaków. Są dwie możliwości załadowania kodu źródłowego: bezpośrednio przez stringa lub przez podanie ścieżki do pliku, który zostanie w całości wczytany. Moduł podczas przetwarzania ciągu wejściowego zapisuje informacje o nowych liniach przez co później można odpytać moduł w celu pobrania wybranej linijki kodu (używane podczas wyświetlania informacji o błędzie).

Analizator leksykalny – Scanner

Lekser będzie na swoim wejściu dostał kolejne znaki z pliku wejściowego od InputManager’a i na tej podstawie będzie tworzył kolejne tokeny, które będą przekazywane do analizatora składniowego. Analizator leksykalny będzie również obsługiwał komentarze (pojedynczej linii “//“ jak również komentarz zakresu “/* (…) */“) i od razu będzie on usuwany. Analiza leksykalna będzie oparta na automacie. Białe znaki będą pomijane na etapie analizy leksykalnej.

Analizator składniowy – Parser

Parser przechowuje w sobie obiekt Scanner’a, aby odpytywać o kolejne tokeny. Na tej podstawie będzie dokonywał rozbioru wyrażeń w celu sprawdzenia czy otrzymane tokeny zgadzają się ze zdefiniowaną wyżej gramatyką programu. Jako rezultat parsowania zostanie zwrócony obiekt typu Program, który jest wierzchołkiem wygenerowanego drzewa rozbioru.

Analizator semantyczny – Environment

Podczas analizy składniowej Parser będzie przekazywał informacje do obiektu Environment takie jak rozpoczęcie/zakończenie bloku czy deklaracja zmiennej. Na tej podstawie obiekt Environment będzie weryfikował poprawność semantyczną:

- czy nie ma odwołań do niezadeklarowanych zmiennych
- czy nie wystąpiła redeklaracja zmiennej w danym bloku
- czy przy odwołaniu do zmiennej, która jest tablicą jest podany indeks
- czy przy odwołaniu do zmiennej niebędącą tablicą nie podano indeksu

Optymalizator – Optimizer
Sama optymalizacja będzie odbywać się w ramach poszczególnych bloków, dlatego sama logika optymalizacji jest zawarta w obiektach, a obiekt Optimizer wywołuje optymalizację na korzeniu drzewa rozbioru, czyli na obiekcie Program zwróconym przez Parser. Każdy blok tworzy swoją własną tablicę symboli z dodatkowymi informacjami o użyciu danej zmiennej w tym bloku i na tej podstawie decyduje się na optymalizacje. Sam algorytm w ramach bloku jest opisany w dalszej części tej dokumentacji. Sama optymalizacja musi się odbyć już po parsowaniu, kiedy jest już stworzony obiekt Program, który zawiera pełne drzewo rozbioru.

Obsługa błędów – ErrorHandler
Każdy z modułów będzie mógł się komunikować z modułem do obsługi błędów. W przypadku napotkania błędu przez któryś z modułów dalsza analiza (tym samym również proces optymalizacji) zostanie przerwany i zostanie wyświetlony stosowny komunikat. Program wykorzystuje klasę CharPos, która przechowuje informację o położeniu w wejściowym kodzie źródłowym, jak również pozwala na pobranie od InputManager'a całej linijki, w której się znajduje. Pod wyświetloną linijką znajduje się strzałka, która wskazuje dokładne położenie znaku, przy którym pojawił się błąd. W jeszcze kolejnej linijce znajduje się informacja o położeniu błędu (numer linijki i numer znaku z tej linijce) oraz treść błędu podana przez moduł, który ten błąd zgłosił.

STRUKTURY DANYCH:

Tablica symboli
Tablica symboli występuje w dwóch miejscach. Pierwsza tablica jest definiowana dla każdego bloku wewnątrz Environment. Każdy element tablicy zawiera w sobie nazwę zmiennej, informację czy jest tablicą czy nie oraz flagę czy zmienna została zadeklarowana w tym bloku.

Druga tablica symboli znajduje się wewnątrz każdego bloku. Przechowuje ona informacje o miejscach (numer statement'u wewnątrz tego bloku) modyfikacji i odczytu tej zmiennej. Jeżeli wewnątrz danego statement'u nastąpiło odczytanie/zapisanie to również jest to uwzględniane w blokach nadrzędnych. Ta tablica jest wykorzystywana do optymalizacji, która odbywa się w ramach pojedynczego bloku.

Drzewo rozbioru
Podczas analizy składniowej są wołane metody odpowiedzialne za parsowanie poszczególnych elementów gramatyki. Każda z tych metod zwraca odpowiedni obiekt, który w praktyce reprezentuje poddrzewo, które zostanie doczepione do drzewa metody nadrzędnej (która wywołała daną metodę). Każdy z obiektów implementuje metodę toString() przez co można wyświetlić cały program lub dowolne pojedyncze poddrzewo.

Tablica zdefiniowanych tokenów
Podczas analizy leksykalnej będzie potrzebna informacja o symbolach zarezerwowanych podczas generowania tokenów przez lekser. Jest ona przechowywana w ReservedTokens.

INNE INFORMACJE:

Parser samemu odpytuje lekser o następne tokeny, a dodatkowo podczas parsowania na bieżąco powiadamia analizator semantyczny o wybranych strukturach. Dzięki temu analiza leksykalna, gramatyczna oraz semantyczna są przeprowadzane równolegle, żeby np. nie weryfikować poprawności leksykalnej i składniowej całego kodu, gdy na samym początku kodu jest błąd semantyczny.

Program

Program będzie aplikacją konsolową napisaną w języku wysokiego poziomu (Java). Przy jego wywołaniu będzie podana ścieżka do pliku wejściowego. Rezultat będzie wyświetlony na standardowym wyjściu.

Testowanie

W celu testowania zostaną napisane testy jednostkowe (korzystające z JUnit) Testy będą sprawdzać np. czy analizator leksykalny prawidłowo wykrywa tokeny na podstawie danego wejściowego ciągu znaków oraz czy analizator składniowy prawidłowo weryfikuje zdefiniowaną gramatykę. Będzie też testowanie poprawności semantycznej oraz poprawności wygenerowanego (zoptymalizowanego) kodu. Testy będą sprawdzać nie tylko poprawny kod, ale również będą weryfikować, czy został wykryty błąd, gdy podano nieprawidłowy kod.

ALGORYTM OPTYMALIZACJI

Każdy blok przechowuje własną tablicę symboli. Każdy symbol zawiera informację czy odpowiadająca zmienna został zadeklarowana w tym bloku oraz miejsca jest zapisu i odczytu w tym bloku. Miejsce zapisu/odczytu jest określone poprzez numer statement'u w ramach tego bloku. Jeżeli dany statement, np. zagnieżdżony blok, modyfikuje lub odczytuje zmienną to zostanie do uwzględnione we wszystkich blokach nadrzędnych, w której ta zmienna istnieje.

1. Dla każdego bloku, w którym może występować zmienna musimy sprawdzić, ile jest tam operacji, które mogą zmodyfikować wartość danej zmiennej, czyli ile jest operacji przypisania (wliczając w to przypisania w blokach zagnieżdżonych). Jeżeli liczba takich operacji jest większa niż jeden, to rozważana zmienna nie może być wyciągnięta poza obecny blok (możliwy jest wyjątek w sytuacji, w której wszystkie operacje przypisania przypisują taką samą stałą lub zmienną, która w danym bloku ma stałą wartość). W przypadku, kiedy jest tylko jedna operacja przypisania, to ta zmienna jest podejrzana o możliwość optymalizacji.

2. W celu sprawdzenia czy rzeczywiście możemy przerzucić daną operację przypisania przed aktualny blok, musimy sprawdzić dwa warunki:

2.1. Czy podczas tego jedynego przypisania jest przypisana wartość stała lub wartość wyrażenia opartego na zmiennych, których wartość jest stała w ramach tego bloku (możemy to odczytać również na podstawie ilości przypisań tej drugiej zmiennej w rozważanym bloku).

2.2. Czy między początkiem bloku, a tym jedynym przypisaniem, jest jakaś operacja, która odwoływałaby się do tej zmiennej. W tym celu w trakcie tworzenia tablicy symboli musimy zapisać w niej również miejsce, w którym jest pierwsze odwołanie do zmiennej w tym bloku oraz miejsce, w którym znajduje się pierwsze przypisanie wartości do tej zmiennej. Mając te dwie dane możemy porównać, która z tym operacji nastąpiła jako pierwsza.

3. Jeżeli oba powyższe warunki są spełnione, to możemy tą jedyną operację przypisania wartości do zmiennej wyciągnąć przed obecny blok.

Optymalizacja odbywa się wewnątrz każdego bloku niezależnie. W celu rozpoczęcia optymalizacji w bloku nadrzędnym wszystkie bloki zagnieżdżone muszą być już zoptymalizowane. Dlatego uruchamiając optymalizacji dane blok musi najpierw wywołać optymalizację dla wszystkich bloków, które są w nim zagnieżdżone.

Przeniesienie niezmiennika poza pętle, zmienia postać kodu tego bloku, co może powodować, że znajdą się kolejne niezmienniki, dlatego optymalizację należy uruchamiać wielokrotnie aż do momentu, gdy nie wyciągnie już niczego przed pętlę.

Optymalizacja odbywa się na blokach, czyli na elementach drzewa rozbioru, czyli wykonując optymalizację modyfikujemy drzewo, z którego można potem wygenerować kod.

Należy uwzględniać, że nie wszystkie operacje, które mają wpływ na optymalizację są bezpośrednio w tym bloku np. w pętli for są trzy argumenty – dwa pierwsze należą do początku bloku, a ostatni należy do końca bloku, więc należy do uwzględnić przy liczeniu miejsc modyfikacji lub odczytu zmiennej.

Przykłady optymalizacji:

Przed:

```
int a = 5;  int b = 10; int c = 20;
int d = 1;  int e = 2;  int f = 3;
int g = 4;  int h = 6;

void main()
{
    for (int i = 0; i < 5; i = i + 1)
    {
        int m = 5;
        h = 10 * a;
        d = b * 7 + 15;
        f = b * 7 + 15;
        c = 50;
        g = 20;
        int j = 10;
        while (j > 10)
        {
            j = j - 1;
            g = 30;
            e = 2 * c;
            for (int k = 0; k < 100; k = k + 1)
            {
                m = (i - 5) * 5;
                f = 20;
                if (f > 15)
                    a = 8;
            }
            f = 1;
        }
    }
}
```

Po:

```
int a = 5;  int b = 10; int c = 20;
int d = 1;  int e = 2;  int f = 3;
int g = 4;  int h = 6;

void main()
{
    d = b * 7 + 15;
    c = 50;
    e = 2 * c;          //adnotacja 1
    for (int i = 0; i < 5; i = i + 1)
    {
        int m = 5;
        h = 10 * a;     //adnotacja 2
        f = b * 7 + 15; //adnotacja 3
        g = 20;
        int j = 10;
        g = 30;
        m = (i - 5) * 5;//adnotacja 4
        while (j > 10)
        {
            j = j - 1;
            f = 20;     //adnotacja 5
            for (int k = 0; k < 100; k = k + 1)
            {
                if (f > 15)
                    a = 8;
            }
            f = 1;
        }
    }
}
```

Adnotacja 1: przeniesione dwa bloki (pętle) wyżej
Adnotacja 2: nie można przenieść, ponieważ a może zmienić wartość
Adnotacja 3: nie można przenieść, ponieważ f może zmienić wartość
Adnotacja 4: przenieśliśmy, ponieważ w obszarze, w którym żyje zmienna m, i ma stałą wartość
Adnotacja 5: przeniesione tylko jeden blok wyżej, bo w obecnym bloku f może zmienić wartość

Uwaga: Kod ma zadanie przedstawić ideę optymalizacji zmiennych. Zakładamy, że w ten kod mogą być „wplecione" inne operacje korzystające z tych zmiennych, dlatego np. nie usuwamy „g = 20;" i nie wyciągamy „g = 30;" przed główną pętlę.

Przykłady optymalizacji z mniejszą liczbą zmiennych:

Przed:	Po:

`float a = 5;`
`void main()`
`{`
`  for (int i = 0; i < 30; i = i + 1)`
`  {`
`  int d = a / 2;`
`  a = 3;`
`  //kod niemodyfikujący a`
`  }`
`}`
	`float a = 5;`
	`void main()`
	`{`
	`  for (int i = 0; i < 30; i = i + 1)`
	`  {`
	`  int d = a / 2;`
	`  a = 3; //nie możemy przenieść zmiennej, bo wcześniej jest odczytywana wartość zmiennej a`
	`  //kod niemodyfikujący a`
	`  }`
	`}`

`float a = 5;`
`void main()`
`{`
`  for (int i = 0; i < 30; i = i + 1)`
`  {`
`  //kod nieodczytujący a`
`  a = 3;`
`  int d = a / 2;`
`  //kod niemodyfikujący a`
`  }`
`}`
	`float a = 5;`
	`void main()`
	`{`
	`  a = 3;`
	`  for (int i = 0; i < 30; i = i + 1)`
	`  {`
	`  //kod nieodczytujący a`
	`  int d = a / 2;`
	`  //kod niemodyfikujący a`
	`  }`
	`}`

`int a = 3;`
`void main()`
`{`
`  for (int i = 0; i < 20; i = i + 1)`
`  {`
`  a = 6;`
`  //kod niemodyfikujący a`
`  a = 4;`
`  }`
`}`
	`int a = 3;`
	`void main()`
	`{`
	`  for (int i = 0; i < 20; i = i + 1)`
	`  {`
	`  a = 6; //nie możemy przenieść zmiennej, bo później jest zmieniana wartość zmiennej a`
	`  //kod niemodyfikujący a`
	`  a = 4;`
	`  }`
	`}`

`int a = 3;`
 	
`void main()`
`{`
`  for (int i = 0; i < 20; i = i + 1)`
`  {`
`  //kod nieodczytujący a`
`  a = 6;`
`  int a = 8;`
`  //kod niemodyfikujący a`
`  a = 4;`
`  }`
`}`
	`int a = 3;`
	
	`void main()`
	`{`
	`  a = 6; //możemy przenieść, bo później jest modyfikowana inna zmienna (o tym samym identyfikatorze)`
	`  for (int i = 0; i < 20; i = i + 1)`
	`  {`
	`  //kod nieodczytujący a`
	`  int a = 8;`
	`  //kod niemodyfikujący a`
	`  a = 4;`
	`  }`
	`}`

`void main()`
`{`
`  for (int i = 0; i < 20; i = i + 1)`
`  {`
`  int k;`
`  for (int j = 0; j < 10; j = j + 1)`
`  {`
`  //kod nieodczytujący k`
`  k = i;`
`  //kod niemodyfikujący k`
`  }`
`  }`
`}`
	`void main()`
	`{`
	`  for (int i = 0; i < 20; i = i + 1)`
	`  {`
	`  int k;`
	`  k = i;`
	`  for (int j = 0; j < 10; j = j + 1)`
	`  {`
	`  //kod nieodczytujący k`
	`  //kod niemodyfikujący k`
	`  }`
	`  }`
	`}`

Przykłady informowania użytkownika o znalezionych błędach:

`void main()`
`{`
`    abc = 3;`
`}`
	`    abc = 3;`
	`    ^`
	`ERROR (2, 4): Using of undeclared variable abc`

`void main()`
`{`
`    int tab[20];`
 	
`    int a = 3 * tab;`
`}`
	`    int a = 3 * tab;`
	`                ^`
	`ERROR (4, 16): Missing index in table variable tab`

`void main()`
`{`
`    float a;`
`    float b;`
`    float c = a b;`
`}`
	`    float c = a b;`
	`                  ^`
	`ERROR (4, 16): Expected SEMICOLON`

`void main()`
`{`
`    float a;`
`    bool b;`
`}`
	`    bool a;`
	`    ^`
	`ERROR (3, 4): Redeclaration of variable a`

Korzystanie z programu:

Program przyjmuje na wejście dwa argumenty: tryb wejścia i samo wejście.

Jeżeli pierwszy argument to *-t*, to program jako drugi argument oczekuje ciągu znaku będącym kodem źródłowym, który ma zostać zoptymalizowany.

Jeżeli pierwszy argument to *-f*, to program jako drugi argument oczekuje ścieżki do pliku, z którego odczyta kod do optymalizacji.

Program wyświetli zoptymalizowany kod na standardowym wyjściu.

Zbudowany plik *.jar* oraz przykładowe kody wejściowe znajdują się w folderze *examples*. Folder zawiera również pliki z wyjściami odpowiadających plików wejściowych.

Przykładowe wywołania programu z folderu *examples*:

```
java -jar kod.jar -f example_in4.txt
```

```
java -jar kod.jar -t "void main() { int x; for (;;) x = 1; }"
```